پیشنهاده طرح پژوهشی برای پروژه پژوهش‌گر مقیم پژوهشکده علوم کامپیوتر IPM

پژوهشگر: مرتضی ذاکری (zakeri@ipm.ir)

آبان ۱۴۰۳

عنوان پروژه

عنوان فارسی: بازآرایی طراحی نرم‌افزار برای بهبود آزمون‌پذیری

عنوان انگلیسی: Refactoring Software Design to Improve Testability

چکیده

آزمون‌پذیری یا قابلیت آزمون یکی از صفات کیفی توسعه نرم‌افزار است که به صورت مستقیم بر کیفیت و تلاش آزمون نرم‌افزار تأثیر می‌گذارد. اغلب وجود نشانه‌های طراحی بد، یعنی بوها و ضدالگوهای طراحی، سبب دشواری آزمون در سطوح مختلف، به‌ویژه یکپارچگی می‌گردد. طراحی نرم‌افزار آزمون‌پذیر نیازمند رعایت اصول و الگوهای طراحی، مانند اصل تک مسؤولیتی یا الگوی تزریق وابستگی است. هدف از این پروژه تشخیص خودکار موقعیت‌های آزمون‌ناپذیر در سطح طراحی نرم‌افزار و اصلاح آنها برای افزایش آزمون‌پذیری است. بهبود آزمون‌پذیری طراحی با فنون *بازآرایی نرم‌افزار* انجام می‌گردد که بایستی بدون تغییر رفتار بیرونی برنامه اعمال شوند. برای سنجش آزمون‌پذیری، روابط متعددی ارائه شده است که هرکدام جنبه‌های مختلفی از این صفت کیفی را در نظر می‌گیرند. لذا رویکرد پیشنهادی باید قادر به بهبود همزمان روابط مختلف باشد. چالش پیش‌روی دیگر، عدم تخریب سایر صفات کیفی در حین بهبود آزمون‌پذیری است؛ زیرا، پژوهش‌ها نشان می‌دهند که صفات کیفی نرم‌افزار متضاد نیز هستند. برای حل این چالش‌ها نوع و ترتیب بازآرایی‌ها در یک فرایند جست‌وجوی چندین هدفه مشخص خواهند شد. راهکار پیشنهادی با تعیین میزان بهبود معیارهای آزمون‌پذیری طراحی بر روی پروژه‌های بزرگ‌مقایس ارزیابی می‌گردد.

۱. مقدمه

آزمون‌پذیری[۱] نرم‌افزار یکی از صفات کیفی داخلی مهم هر سیستم نرم‌افزاری است که به سهولت آزمون نرم‌افزار از نظر یافتن خطا اشاره دارد [۱]. آزمون‌پذیری، شناسایی و رفع مشکلات احتمالی در نرم‌افزار را برای توسعه‌دهندگان آسان می‌کند که در نهایت می‌تواند کیفیت و قابلیت اطمینان نرم‌افزار را برای کاربران و برنامه نویسان بهبود بخشد [۲]. فقدان آزمون‌پذیری از نظر زمانی، مالی و حتی حیاتی، آسیب‌های جبران‌ناپذیری برای کاربر یا توسعه‌دهنده سیستم، می‌تواند ایجاد کند [۳]. در سطح طراحی آزمون‌پذیری ناظر بر چیدمان درست مؤلفه‌های برنامه در راستای تسهیل روند آزمون‌های واحد و یکپارچگی و افزایش اثربخشی آنها است.

یکی از راه‌های موثر برای بهبود آزمون‌پذیری، حذف بوهای کد است که آن را کاهش می‌دهند. توسعه دهندگان از روش‌های بازآرایی به طور خاص برای آزمون‌پذیری استفاده می‌کنند. بازآرایی برای آزمون‌پذیری[۲] شامل اصلاح ساختار کد و طراحی بدون تغییر رفتار بیرونی آن است که در نتیجه کار تولید و اجرای آزمون‌ها را آسان‌تر می‌کند [۴]. به‌نظر می‌رسد که برخی از بازآرایی‌های متداول، بوهای طراحی و کد کاهش‌دهنده آزمون‌پذیری را حذف می‌کنند. از جمله عملیات بازآرایی که مستلزم خودکارسازی در راستای افزایش آزمون‌پذیری هستند می‌توان به استخراج متد [۳][۵]، استخراج کلاس [۴][۶]، انتقال متد [۵][۷]، انتقال فیلد [۶][۸]، کلاس درون‌خطی [۷][۸]، کپسوله کردن فیلد [۸][۸]، بالا کشیدن متد [۹][۸] و جایگزینی پارامتر با پرس‌وجو [۱۰][۸] اشاره کرد.

برای محاسبه آزمون‌پذیری روابط متعددی وجود دارد و استفاده از این روابط مختلف به‌عنوان اهداف بازآرایی خودکار، یک رویکرد قدرتمندتر را برای افزایش کیفیت نرم‌افزار نسبت به حالت تک هدفی، معرفی می‌کند. این روش با در نظر گرفتن جنبه‌های مختلف کد منبع، که نیاز به الگوریتم‌های چندین هدفه دارد، امکان ارزیابی جامع‌تری از آزمون‌پذیری را فراهم می‌کند. کارهای موجود در بازآرایی خودکار طراحی نرم‌افزار، این مهم را در نظر نگرفته و حداکثر یک جنبه از آزمون‌پذیری را به‌عنوان هدف استفاده می‌کنند. به عنوان مثال، Morales و همکاران [۹]، یک رویکرد چند هدفه مبتنی بر جستجوی جدید ([۱۰] MOCell،[۱۱] SPEA2 ،[۱۲] NSGA-II) برای بازآرایی، درجهت حذف پنج ضد الگوی رایج و به حداقل رساندن تلاش آزمون پیشنهاد کردند.

Morales و همکاران [۹]، آزمون‌پذیری را تنها با استفاده از MaDUM محاسبه می‌کنند، که تلاش آزمون را بر اساس استراتژی تقسیم و غلبه بر روی آزمون واحد اعمال می‌شود، تخصیص می‌دهد. Zakeri و همکاران [۱۳]، نشان می‌دهند که با تغییر خودکار کلاس‌های بدبو جاوا برای بهبود معیارهای نرم‌افزاری تأثیرگذار، آزمون‌پذیری این کلاس‌ها را می‌توان به طور متوسط تا ۸۶.۸۷ درصد افزایش داد. آنها آزمون‌پذیری نرم‌افزار بر اساس پوشش و تعداد موارد آزمون ارائه شده توسط مجموعه آزمون با در نظر

---

[۱] Testability

[۲] Refactoring for testability

[۳] Extract Method

[۴] Extract Class

[۵] Move Method

[۶] Move Field

[۷] Inline Class

[۸] Encapsulate Field

[۹] Pull-Up Method

[۱۰] Replace Parameter with Query

گرفتن بودجه آزمون محاسبه می‌کنند ولی اثر بازآرایی‌های چندتایی (یک توالی از بازآرایی‌ها) و مرکب دیده نشده است. همچنین، Hays و همکاران [۱۴]، آزمون‌پذیری را به عنوان معیاری برای سنجش سهولت آزمون یک گره خاص در نمودار جریان کنترل[۱] یک تابع تعریف می‌کنند. Suri و همکاران [۱۵]، آزمون‌پذیری را از نظر قابل فهم بودن و پیچیدگی کدهای آزمون بررسی کرده‌اند. برای بهبود آزمون‌پذیری از جنبه‌های مختلف نیاز به الگوریتم‌های چندین هدفه داریم که بتواند بهترین ترکیب از بازآرایی‌ها را پیدا کرده، به‌نحوی که آزمون‌پذیری بیشینه شود. الگوریتم‌های بهینه‌سازی تکاملی مانند RVEA [۱۶]، [۱۷] SPARVEA، [۱۸] C-TAEA-II،[۱۹] NAEMO، [۲۰] U-NSGA-III و [۲۱] IBEA-FC، با توجه به بررسی‌های انجام شده نسبت به الگوریتم‌های که قبلاً استفاده شده‌اند، بهتر هستند.

پژوهش‌های پیشین برای بهبود آزمون‌پذیری طراحی به‌صورت خودکار، توالی از بازآرایی‌ها را مشخص نکرده‌اند که از زوایای مختلف آزمون‌پذیری طراحی نرم‌افزار را بهبود بخشند. در طرح پیشنهادی، با هدف یافتن یک توالی بهینه برای خودکارسازی بازآرایی و بهبود آزمون‌پذیری طراحی، ابتدا بازآرایی‌ها بر اساس موقعیت‌های کاهش‌دهنده آزمون‌پذیری شناسایی و اعمال آنها خودکارسازی می‌شود. در مرحله دوم، روابط مختلف آزمون‌پذیری طراحی نرم‌افزار را پیاده‌سازی شده و سپس در مرحله سوم، همبستگی بین روابط مختلف آزمون‌پذیری را بدست می‌آوریم تا بتوان معیارهای که با یکدیگر همبستگی کمتری را دارند به عنوان شاخص‌های اصلی آزمون‌پذیری طراحی معرفی کرد. مرحله چهارم، شامل پیاده‌سازی الگوریتم بهینه‌سازی چندهدفه برای بدست آوردن توالی از بازآرایی‌ها به منظور بیشینه‌سازی آزمون‌پذیری و در عین حال عدم تخریب صفات کیفی است. در نهایت، میزان تأثیر هریک از بازآرایی‌ها، بر آزمون‌پذیری از جنبه‌های مختلف ارزیابی می‌شوند.

۲. کارهای مرتبط

توسعه‌دهندگان می‌توانند با *بازآرایی برای آزمون‌پذیری*[۲] [۴]، تلاش آزمون را کاهش دهند، قابلیت نگهداری کد را افزایش دهند و کیفیت کلی محصول نرم‌افزاری را افزایش دهند [۹], [۱۳], [۲۲]. Zakeri و همکاران [۱۳]، بازآرایی برای بهبود آزمون‌پذیری انجام دادند, انها یک رویکرد جدید برای تخمین و بهبود آزمون‌پذیری در پروژه های نرم‌افزاری بر اساس پوشش و تعداد موارد آزمون ارائه شده توسط مجموعه آزمون، با در نظر گرفتن بودجه آزمون اندازه گیری می‌شود, معرفی کردند. این مطالعه ۲۳۰۰۰ کلاس از ۱۱۰ پروژه جاوا را با معیار آزمون‌پذیری برچسب‌گذاری می‌کند، آن‌ها را با استفاده از ۲۶۲ معیار بردار می‌کند، الگوریتم‌های یادگیری ماشینی [۳]نظارت‌شده، به‌ویژه رگرسیون[۴] را برای پیش‌بینی آزمون‌پذیری بر اساس معیارهای کد منبع اعمال می‌کند. مدل‌های رگرسیون به R^2 برابر با ۰.۶۸ و میانگین مربعات خطای ۰.۰۳ دست می‌یابند که نشان دهنده مناسب بودن این عمل است. این پژوهش، ۱۵ معیار نرم‌افزاری را شناسایی می‌کند که به طور قابل توجهی بر پیش بینی آزمون‌پذیری از طریق تجزیه و تحلیل اهمیت ویژگی در مدل آموخته شده تأثیر می‌گذارد. بازآرایی خودکار کلاس‌هایی که این معیارهای تأثیرگذار را هدف قرار می‌دهند، می‌تواند آزمون‌پذیری را به طور متوسط ۸۶.۸۷٪ بهبود بخشد. این مطالعه تعداد محدودی بازآرایی را شامل می‌شود و ترتیب بازآرایی‌ها اهمیت ندارد و توالی از بازآرایی‌ها را که باعث افزایش آزمون‌پذیری می‌شود را در اختیار ما نمی‌گذارد.

Cinnéide و همکاران [۴]، پتانسیل افزایش آزمون‌پذیری برنامه را از طریق بازآرایی خودکار، به ویژه با استفاده از معیار انسجام[۵] کلاس مبتنی بر شباهت طراحی سطح پایین (LSCC) [۲۳] بررسی کردند. برای ارزیابی کار خود، آنها یک شبه آزمون را انجام

---

۱ Control flow

۲ Refactoring for testability

۳ Machine learning

۴ Regression models

۵ Cohesion

دادند که در آن یک برنامه کوچک با انسجام ضعیف به طور خودکار با استفاده از معیار LSCC بازآرایی شد. با کمال تعجب، نتایج آزمون ساختگی نشان داد که آنها نتوانستند توالی از بازآرایی‌ها را پیدا کنند که آزمون‌پذیری را افزایش دهد. با این حال، تحقیقات بعدی با موفقیت توالی از بازآرایی‌ها را شناسایی کرده‌اند که به طور قابل توجهی تلاش آزمون را کاهش داده است. Sabane و همکاران [۲۲]، توالی از بازآرایی‌ها را پیشنهاد کردند که به طور خاص برای کلاس‌های درگیر در ضدالگوها قابل استفاده است، با هدف کاهش هزینه‌های آزمون انجام می‌شوند. این بازآرایی‌ها به صورت دستی انجام شد.

در مقابل، Morales و همکاران [۲۴]، یک الگوریتم جستجوی بازآرایی خودکار را پیشنهاد کرد. این مطالعه بر روی تأثیر ضدالگوها بر تلاش‌های آزمون متمرکز بود. یک رویکرد چند هدفه مبتنی بر جستجو برای خودکارسازی بازآرایی و افزایش کیفیت طراحی و در عین حال به حداقل رساندن تلاش آزمون معرفی شد. مطالعات [۹], [۲۲] تلاش آزمون را با استفاده از MaDUM محاسبه کردند. با این حال، ما متوجه شدیم که هیچ مطالعه‌ای هنوز توالی از بازآرایی‌ها را منتشر نکرده است که آزمون‌پذیری را از جنبه‌های مختلف بهبود دهد.

۳. روش پیشنهادی

این پژوهش با استفاده از روش مطالعه سیستماتیک، در چهار گام اصلی به شرح زیر اجرا می‌شود.

گام ۱. مطالعه مروری: ابتدا مرور سیستماتیک ادبیات بر روی روش‌های بازآرایی خودکار طراحی نرم‌افزار برای بهبود آزمون‌پذیری و پیش‌بینی بازآرایی انجام می‌شود. هدف از این مطالعه، مشخص کردن کارهای لبه پژوهش، خلأ تحقیقاتی و مسائل باز در حوزه بازآرایی خودکار نرم‌افزار و بهبود صفات کیفی است. همچنین، زیرساخت ریاضی مدل‌های یادگیری ماشین و مدل‌های تبدیل برنامه، بررسی و فرمول‌بندی می‌شود. با توجه به بررسی‌های اولیه انجام شده لازمه ایجاد راهکار کارا و اثر بخش برای تحلیل ایستا و تبدیل شکل برنامه در سطح طراحی به نحوی که تغییرات به سطح کد قابل انتقال باشد، استفاده از روش‌های پیشرفته ریاضیاتی به خصوص نمایش گرافی برنامه‌ها است. به همین دلیل، در این پژوهش از فنون مبتنی بر گراف بهره‌گیری می‌گردد تا روش پیشنهادی مبنای ریاضی خوبی داشته باشد و بتوان نتایج آن را به راحتی تفسیر کرده و تعمیم داد.

گام ۲. جمع‌آوری مجموعه داده و مجموعه محک: در این گام، مجموعه داده‌ای از انواع بوهای کد و طراحی نرم‌افزار به همراه مصنوعات لازم برای تشخیص آنها یعنی کد منبع و نمودارهای طراحی تهیه می‌گردد. هدف ایجاد و ارزیابی روش تشخیص خودکار موقعیت‌های آزمون‌ناپذیر برای استفاده در فرایند بازآرایی است. همچنین، تعدادی از پروژه‌های متن‌باز که موقعیت‌های بازآرایی متفاوتی دارند، به‌عنوان مجموعه محک، برای آزمون و ارزیابی روش بازآرایی ارائه شده جمع‌آوری می‌گردد. با استفاده از ابزارهایی مانند JDeodorant می‌توان بازآرایی‌های انجام شده در کدهای مختلف را مشخص و مستند کرد. برای هر طراحی نرم‌افزار آزمون‌پذیری آن با استفاده از معیارهای سنجش می‌شود.

گام ۳. توسعه روش پیشنهادی: در مرحله ۳، الگوریتم‌های خودکارسازی برای تعدادی از بازآرایی‌های مهم مشاهده شده در مجموعه محک، پیاده‌سازی می‌شود، به نحوی که بازآرایی‌ها در سطح کد منبع قابل اعمال باشند. برای اطمینان از درستی فرایند تبدیل برنامه ، بررسی پیش‌شرط‌ها و پس‌شرط‌ها و اعمال بازآرایی بر روی درخت تجزیه نحوی برنامه، با در نظر گرفتن گراف‌های وابستگی برنامه، فراخوانی توابع و جریان کنترل و با استفاده از فنون کامپایلری صورت می‌پذیرد. درخت نحوی تغییر داده شده سپس به کد منبع تبدیل می‌شود. در این مرحله همچنین، روش‌هایی برای کمی‌سازی صفات کیفی مختلف از جمله قابلیت استفاده مجدد و انواع روابط آزمون‌پذیری ارائه شده و آنها را به‌عنوان توابع هدف در الگوریتم بازآرایی مبتنی بر جست‌وجو استفاده خواهیم کرد.

گام ۴. ارزیابی روش پیشنهادی و مقایسه با کارهای مرتبط: در پایان روش ارائه شده برای بهبود آزمون‌پذیری طراحی با استفاده از بازآرایی خودکار نرم‌افزار را بر روی تعدادی از پروژه‌های مجموعه محک جمع‌آوری شده ارزیابی می‌کنیم. همچنین، در صورت امکان این روش پیشنهادی با روش‌های قبلی مقایسه شده تا کارایی و اثر بخشی آن بهتر تبیین گردد.

۴. زمان‌بندی و منابع

زمان‌بندی و منابع پیشنهادی برای این طرح پژوهشی، مطابق جدول ۱ است. همچنین، جدول ۲ زمان‌بندی طرح را در قالب نمودار گانت نشان می‌دهد.

جدول ۱. زمان‌بندی و منابع پیشنهادی

شماره مرحله / فعالیت	نام مرحله/ فعالیت	مرحله/ فعالیت پیش‌نیاز	درصد وزنی به کل پروژه	نام منابع مورد استفاده	مدت زمان اجرا (ماه)
۱	مرور ادبیات و منابع	-	۱۰٪	اینترنت، کامپیوتر شخصی	۲
۲	طراحی و معماری روش پیشنهادی	۱	۱۰٪	اینترنت، کامپیوتر شخصی	۱
۳	پیاده‌سازی الگوریتم‌های بازآرایی خودکار	۱و۲	۲۵٪	اینترنت، کامپیوتر workstation	۳
۴	پیاده‌سازی الگوریتم‌های سنجش آزمون‌پذیری	۱و۲	۱۰٪	اینترنت، کامپیوتر workstation	۱
۵	پیاده‌سازی الگوریتم بهینه‌سازی چند هدفه	۳و۴	۱۰٪	اینترنت، کامپیوتر workstation	۱
۶	ارزیابی روش پیشنهادی	۵	۱۰٪	اینترنت، کامپیوتر workstation	۱
۷	مستندسازی پژوهش، نوشتن و ارسال مقاله‌های علمی	۱و۵و ۶	۲۵٪	اینترنت، کامپیوتر شخصی	۳

جدول ۲. گانت پروژه

شماره مرحله	بازه زمانی (ماه)											
	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲
۱												
۲												
۳												
۴												
۵												
۶												
۷												

۵. نتیجه‌گیری

این پژوهش مفاهیم بنیادی و کاربردی در حوزه مهندسی نرم‌افزار خودکار و هوشمند را ارتقاء می‌دهد. پیوند میان کامپایلرها و مهندسی نرم‌افزار ناشناخته باقی‌مانده است و در حالی که بسیاری وظایف مهندسی نرم‌افزار توسط کامپایلرها قابلیت خودکارسازی دارند. این مهم که بازآرایی خودکار تنها با تکیه بر فنون کامپایلری در هنگام تبدیل برنامه، قابل انجام به صورت صحیح است، معمولاً نادیده گرفته می‌شود. از منظر بنیادی، طی این پژوهش، تمرکز بیشتری بر بهینه‌سازی در کد منبع سطح بالا و طراحی به جای بهینه‌سازی کد یا نمایش میانی، معطوف می‌شود؛ زیرا، برخلاف کد میانی، تغییرات در سطح کد منبع توسط برنامه‌نویسان قابل مشاهده و ملموس است و به کاهش مستقیم قرض فنی و هزینه‌های نگهداشت نرم‌افزار کمک می‌کند. از نقطه نظر فنی، بهنیه‌سازی به جبهه‌جلویی کامپایلرها نیز وارد می‌شود. در نتیجه، توسعه و استفاده از کامپایلرهای جعبه‌سفید برای پشتیبانی از سنجش و بهبود کیفیت خودکار برنامه‌ها نیز در کنار فنون هوش مصنوعی و واکاوش نرم‌افزار، محبوب‌تر می‌گردد.

منابع

[1] V. Terragni, P. Salza, and M. Pezzè, “Measuring Software Testability Modulo Test Quality,” in *Proceedings of the 28th International Conference on Program Comprehension*, New York, NY, USA: ACM, Jul. 2020, pp. 241–251. doi: 10.1145/3387904.3389273.

[2] R. Sharma and A. Saha, “A Systematic Review of Software Testability Measurement Techniques,” in *2018 International Conference on Computing, Power and Communication Technologies (GUCON)*, IEEE, Sep. 2018, pp. 299–303. doi: 10.1109/GUCON.2018.8675006.

[3] V. Garousi, M. Felderer, and F. N. Kılıçaslan, “A survey on software testability,” *Inf Softw Technol*, vol. 108, pp. 35–64, Apr. 2019, doi: 10.1016/j.infsof.2018.12.003.

[4] M. Ó. Cinnéide, D. Boyle, and I. H. Moghadam, “Automated Refactoring for Testability,” in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, IEEE, Mar. 2011, pp. 437–443. doi: 10.1109/ICSTW.2011.23.

[5] M. Shahidi, M. Ashtiani, and M. Zakeri-Nasrabadi, “An automated extract method refactoring approach to correct the long method code smell,” *Journal of Systems and Software*, vol. 187, p. 111221, May 2022, doi: 10.1016/j.jss.2022.111221.

[6] M. Alzahrani, “Extract Class Refactoring Based on Cohesion and Coupling: A Greedy Approach,” *Computers*, vol. 11, no. 8, p. 123, Aug. 2022, doi: 10.3390/computers11080123.

[7] J. Al Dallal, “Predicting move method refactoring opportunities in object-oriented code,” *Inf Softw Technol*, vol. 92, pp. 105–120, Dec. 2017, doi: 10.1016/j.infsof.2017.07.013.

[8] A. Almogahed and M. Omar, “Refactoring Techniques for Improving Software Quality: Practitioners’ Perspectives,” *Journal of Information and Communication Technology*, vol. 20, 2021, doi: 10.32890/jict2021.20.4.3.

[9] R. Morales, A. Sabane, P. Musavi, F. Khomh, F. Chicano, and G. Antoniol, “Finding the Best Compromise Between Design Quality and Testing Effort During Refactoring,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, Mar. 2016, pp. 24–35. doi: 10.1109/SANER.2016.23.

[10] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, “MOCell: A cellular genetic algorithm for multiobjective optimization,” *International Journal of Intelligent Systems*, vol. 24, no. 7, pp. 726–746, Jul. 2009, doi: 10.1002/int.20358.

[11] V. J. Amuso and J. Enslin, “The Strength Pareto Evolutionary Algorithm 2 (SPEA2) applied to simultaneous multi- mission waveform design,” in *2007 International Waveform Diversity and Design Conference*, IEEE, Jun. 2007, pp. 407–417. doi: 10.1109/WDDC.2007.4339452.

[12] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr. 2002, doi: 10.1109/4235.996017.

[13] M. Zakeri-Nasrabadi and S. Parsa, “An ensemble meta-estimator to predict source code testability,” *Appl Soft Comput*, vol. 129, p. 109562, Nov. 2022, doi: 10.1016/j.asoc.2022.109562.

[14] M. Hays and J. Hayes, “The effect of testability on fault proneness a case study of the Apache HTTP Server,” in *Proceedings - 23rd IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2012*, 2012, pp. 153–158. doi: 10.1109/ISSREW.2012.48.

[15] “Object Oriented Software Testability (OOST) Metrics Analysis,” *International Journal of Computer Applications Technology and Research*, pp. 359–367, Apr. 2023, doi: 10.7753/IJCATR0405.1006.

[16] R. Cheng, Y. Jin, M. Olhofer, and B. Sendhoff, “A Reference Vector Guided Evolutionary Algorithm for Many-Objective Optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 5, pp. 773–791, Oct. 2016, doi: 10.1109/TEVC.2016.2519378.

[17] W. Li, Y. Chen, Y. Dong, and Y. Huang, “A solution potential-based adaptation reference vector evolutionary algorithm for many-objective optimization,” *Swarm Evol Comput*, vol. 84, p. 101451, Feb. 2024, doi: 10.1016/j.swevo.2023.101451.

[18] K. Li, R. Chen, G. Fu, and X. Yao, “Two-Archive Evolutionary Algorithm for Constrained Multiobjective Optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 2, pp. 303–315, Apr. 2019, doi: 10.1109/TEVC.2018.2855411.

[19] R. Sengupta, M. Pal, S. Saha, and S. Bandyopadhyay, “NAEMO: Neighborhood-sensitive archived evolutionary many-objective optimization algorithm,” *Swarm Evol Comput*, vol. 46, pp. 201–218, May 2019, doi: 10.1016/j.swevo.2018.12.002.

[20] H. Seada and K. Deb, “U-NSGA-III : A Unified Evolutionary Algorithm for Single , Multiple , and Many-Objective Optimization,” 2014.

[21] M. Basseur and E. K. Burke, “Indicator-based multi-objective local search,” in *2007 IEEE Congress on Evolutionary Computation*, IEEE, Sep. 2007, pp. 3100–3107. doi: 10.1109/CEC.2007.4424867.

[22] A. Sabane, M. Di Penta, G. Antoniol, and Y. Gueheneuc, “A Study on the Relation between Antipatterns and the Cost of Class Unit Testing,” in *2013 17th European Conference on Software Maintenance and Reengineering*, IEEE, Mar. 2013, pp. 167–176. doi: 10.1109/CSMR.2013.26.

[23] J. Al Dallal and L. C. Briand, “An object-oriented high-level design-based class cohesion metric,” *Inf Softw Technol*, vol. 52, no. 12, pp. 1346–1361, Dec. 2010, doi: 10.1016/j.infsof.2010.08.006.

[24] D. Maiorca, I. Corona, and G. Giacinto, “Looking at the bag is not enough to find the bomb,” in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, New York, NY, USA: ACM, May 2013, pp. 119–130. doi: 10.1145/2484313.2484327.